TESTES DE SOFTWARE

PÓS-GRADUAÇÃO *LATO SENSU* EM ARQUITETURA DE SOFTWARE

DISCIPLINA: Qualidade em Arquitetura de Software

Professor: Clávison M. Zapelini

Testar ≠ Depurar

Simplificando

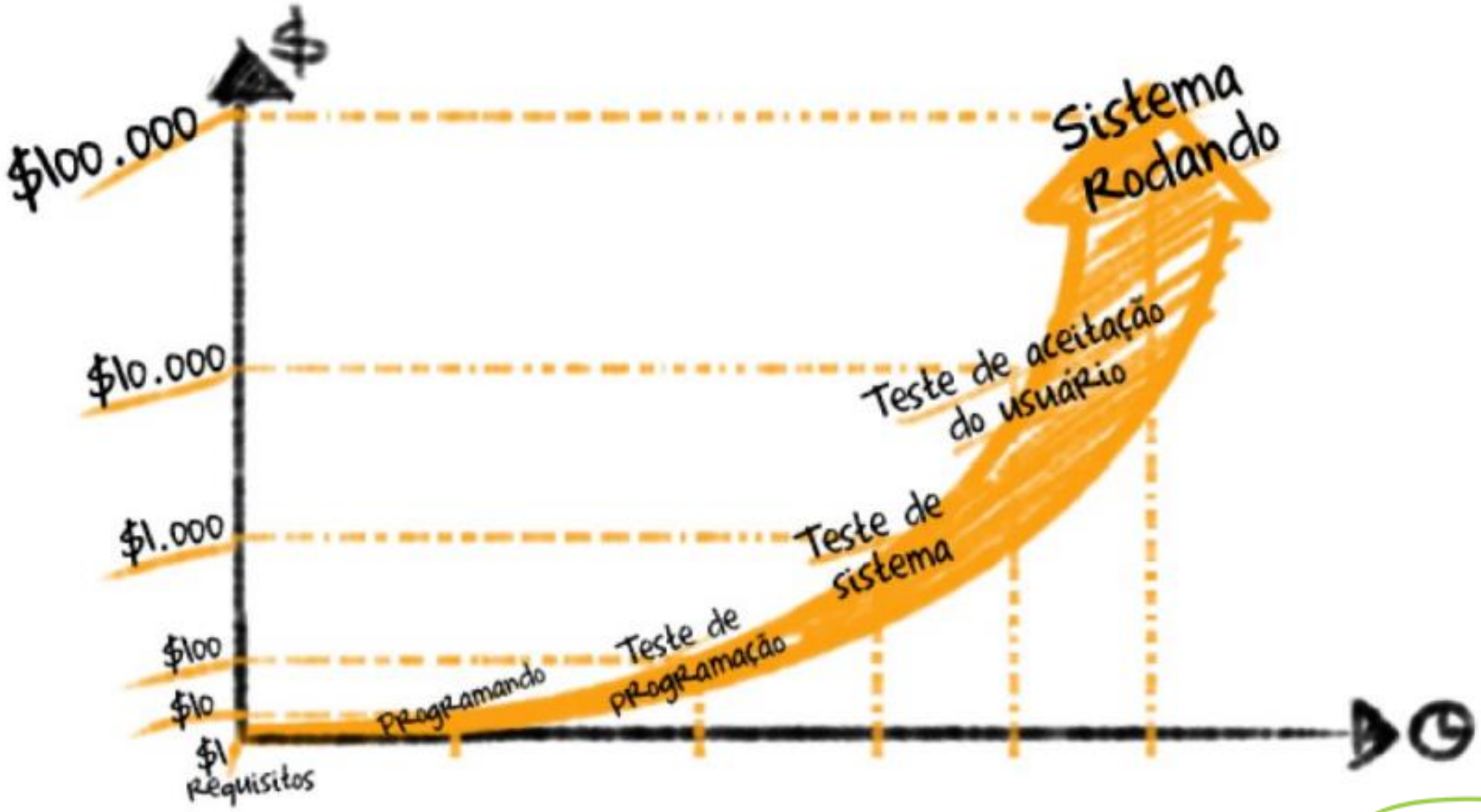
- Depurar - o que se faz quando se sabe que o programa não funciona;
- Teste - tentativas sistemáticas de encontrar erros em programa que você “acha” que está funcionando.

“Testes podem mostrar a presença de erros, não a sua ausência.”

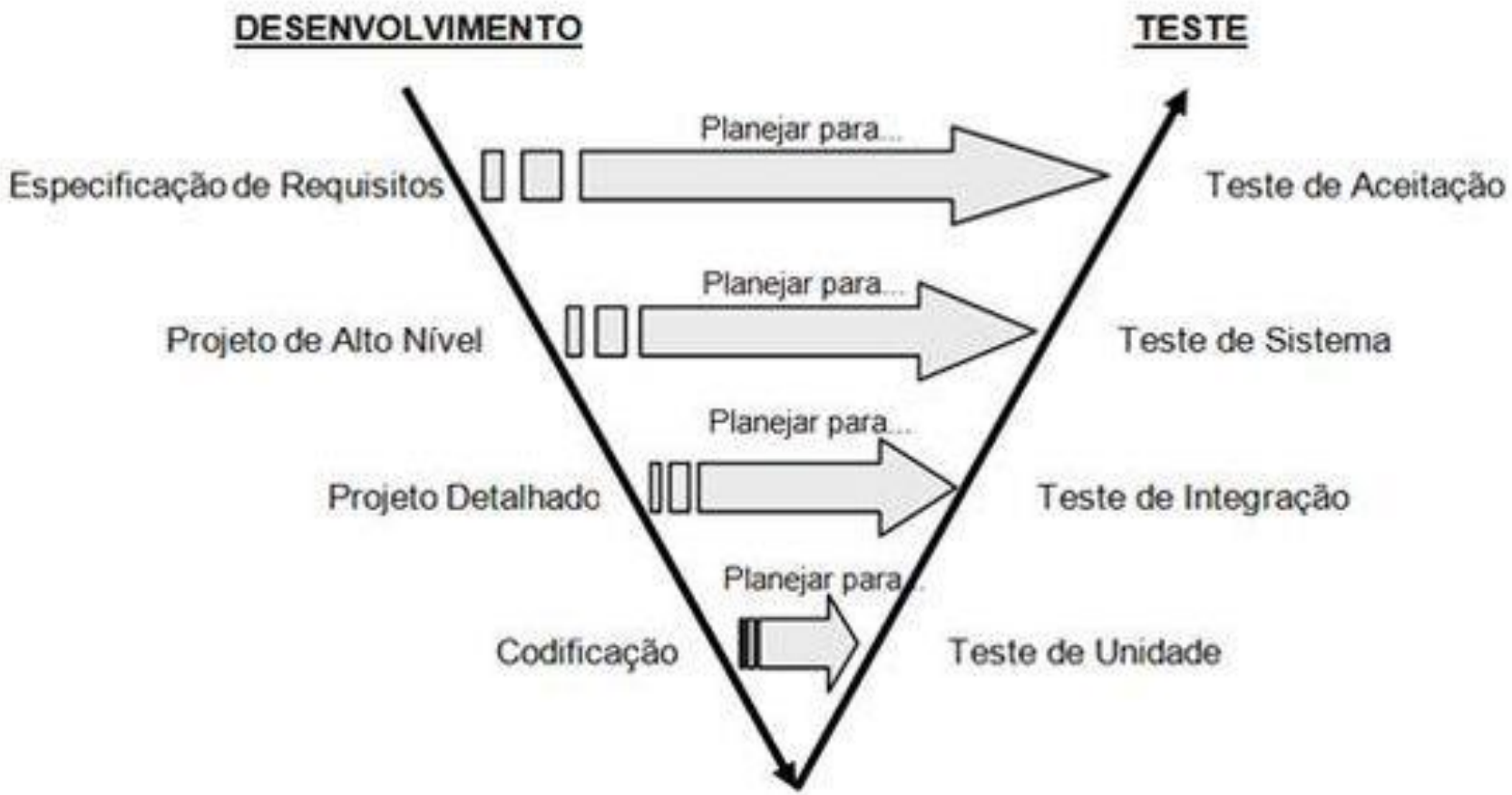
unibave

Quais os benefícios de usar testes automatizados?

- **Redução de custos:** inicialmente o investimento pode ser maior, mas, após a implementação, torna-se menor.
- **Eficiência nas operações:** fatores contextuais dos testes manuais.
- **Aumento da produtividade:** retorno mais rápido das falhas aos desenvolvedores.
- **Redução de erros no testes:** execução dos roteiros de testes por pessoas pode gerar erros no processo.

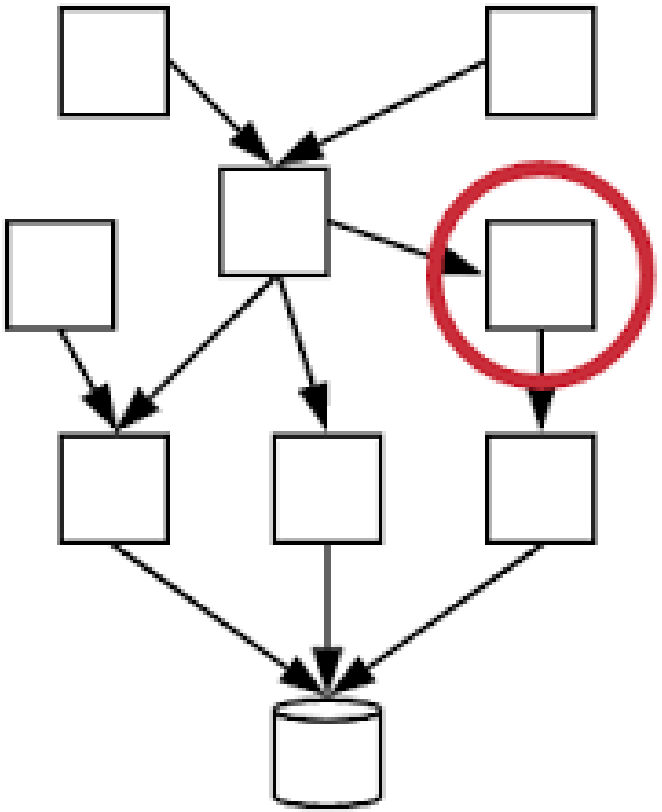

$
$100.000
$10.000
$1.000
$100
$10
$1
Requisitos
Programando
Teste de programação
Teste de sistema
Teste de aceitação do usuário
Sistema Rodando

TIPOS DE TESTES



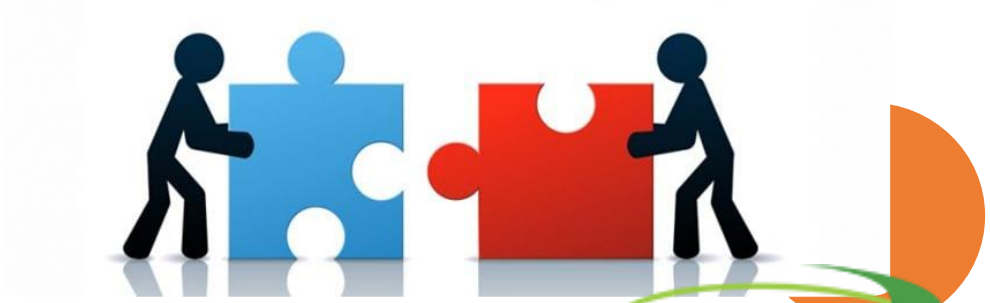
Teste de unidade:

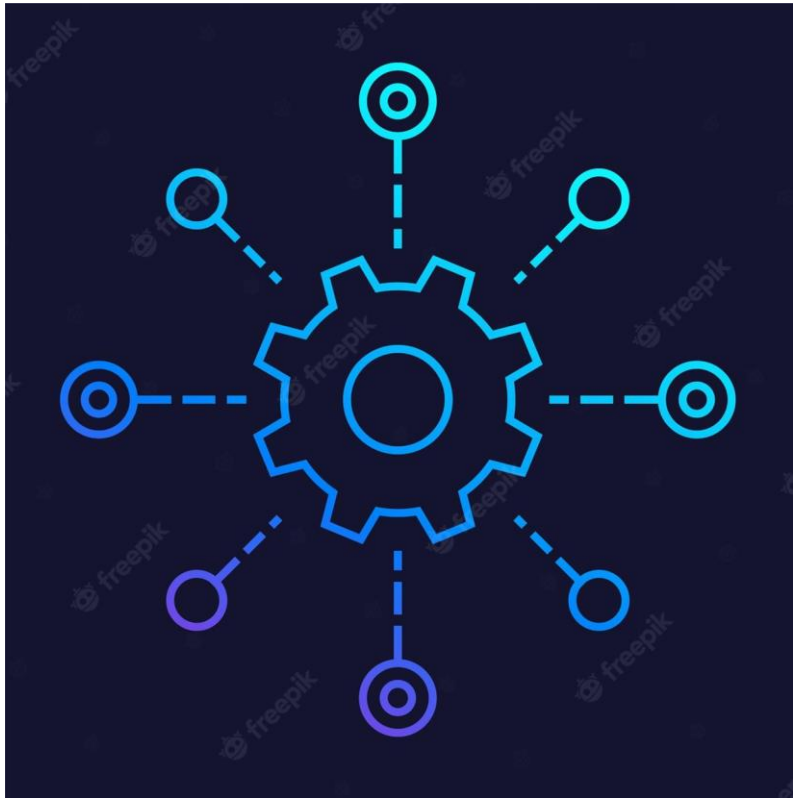
•Nesse tipo de teste, as unidades individuais de código são testadas de forma isolada para verificar se funcionam corretamente. As unidades podem ser funções, métodos ou mesmo classes inteiras. É geralmente realizado pelos desenvolvedores e é essencial para garantir que cada componente do software esteja funcionando corretamente antes de ser integrado ao sistema.



Teste de integração:

•Esse tipo de teste verifica se os diferentes componentes do software funcionam corretamente em conjunto. Ele testa a interação entre módulos, subsistemas ou até mesmo sistemas externos, garantindo que as interfaces entre eles estejam funcionando adequadamente. O objetivo é identificar problemas de comunicação, compatibilidade e dependências entre os componentes.





Teste de sistema:

•Esse teste avalia o sistema de software como um todo para verificar se atende aos requisitos especificados. Ele verifica se o sistema se comporta conforme o esperado em diferentes cenários e condições. O teste de sistema pode incluir diferentes aspectos, como funcionalidade, desempenho, segurança e usabilidade, dependendo dos requisitos do sistema.

Teste de aceitação

•Também conhecido como teste de validação, esse tipo de teste é realizado para verificar se o software está pronto para ser aceito pelo cliente ou usuário final. Ele verifica se o software atende aos critérios de aceitação definidos e se está de acordo com as expectativas do cliente. O teste de aceitação pode incluir testes funcionais, de usabilidade, de desempenho e de conformidade, entre outros.

Teste de regressão:

•Esse teste é realizado para garantir que as alterações recentes no software não tenham introduzido novos defeitos ou quebrado funcionalidades existentes. Ele envolve a reexecução de testes previamente executados para garantir a estabilidade do sistema. A automação de testes de regressão é comum para economizar tempo e garantir uma cobertura abrangente durante as iterações de desenvolvimento.

Teste de desempenho:

•Esse tipo de teste avalia o desempenho do software em diferentes condições de carga e uso. Ele verifica se o sistema é capaz de lidar com volumes esperados de transações, usuários simultâneos e requisitos de tempo de resposta. O teste de desempenho pode identificar gargalos, vazamentos de memória, problemas de escalabilidade e outros aspectos relacionados ao desempenho do sistema.

Teste de segurança:

•Esse teste é realizado para identificar vulnerabilidades de segurança no software. Ele verifica se o sistema é resistente a ataques, como invasões, explorações de falhas de segurança e roubo de informações. O teste de segurança pode incluir análise de código, testes de penetração, simulação de ataques e revisões de configuração do sistema.

INDEPENDENTE DO TIPO

Teste enquanto você escreve código

- Se possível escreva os testes antes mesmo de escrever o código.
  Uma das técnicas de XP.

- Quanto antes for encontrado o erro melhor!

INDEPENDENTE DO TIPO

Teste o código em seus limites

- Para cada pequeno trecho de código (um laço ou condição, por exemplo) verifique o seu bom funcionamento;
- Tente uma entrada vazia, um único item, um vetor cheio, etc.

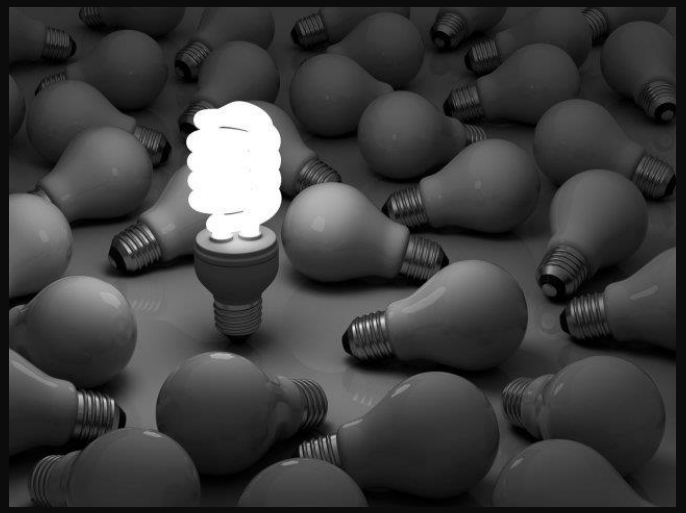
INDEPENDENTE DO TIPO

Teste de pré e pós condições

- Verificar certas propriedades antes e depois de trechos de código.

Faça testes independentes.

- Criar uma bateria de testes onde um depende do outro não testa o que realmente deve ser testado.



unibave

First

Fast: quanto mais rápido melhor.

Independent: não pode depender de outro, deve poder ser executado independente da ordem da execução.

Repeatable: independente da quantidade de vezes que ele for executado, a saída dele deve ser a mesma.

Self-validating: cada teste precisa ter pelo menos uma assertiva para validar a execução.

Thorough/Timely: os testes precisam realizar uma cobertura de forma inteligente, buscando identificar todos os possíveis cenários, para garantir que o software funcionará em todos eles. Também precisam ser feitos no momento correto, a criação dos testes ajudam a deixar o código mais limpo, então eles devem ser feitos antes de ir para produção.

E quando um erro é encontrado?

- Se não foi encontrado por um teste, faça um teste que o provoque.

Para começar

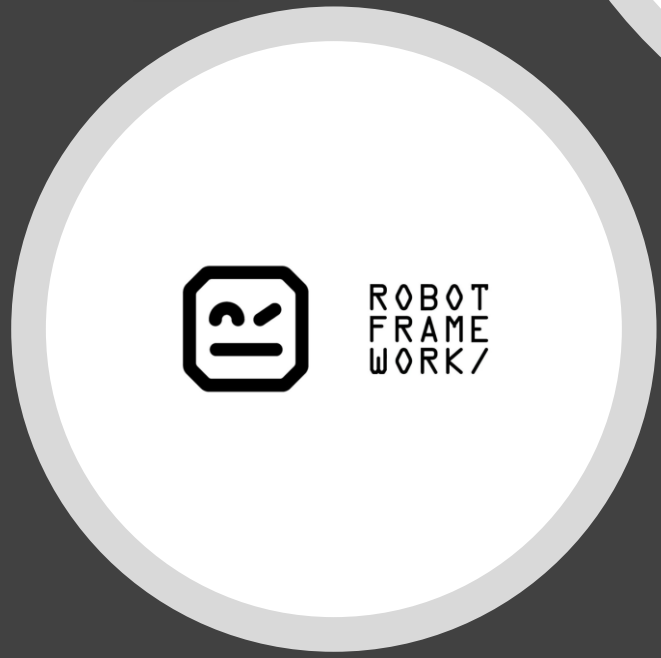
- **Planejamento:** o script deve ser o menor esforço no processo de automação dos testes. Inicie pelo planejamento do seu ciclo.

- **Ambiente**: é necessário criar um ambiente com características iguais aonde o sistema é executado.

- **Ferramentas**: a escolha das ferramentas e dos Frameworks define o sucesso ou o fracasso do projeto. A escolha errada pode acarretar lentidão e ineficiência dos testes.

- **Documentação**: manter uma boa documentação dos testes automatizados é extremamente importante.

•Ferramentas

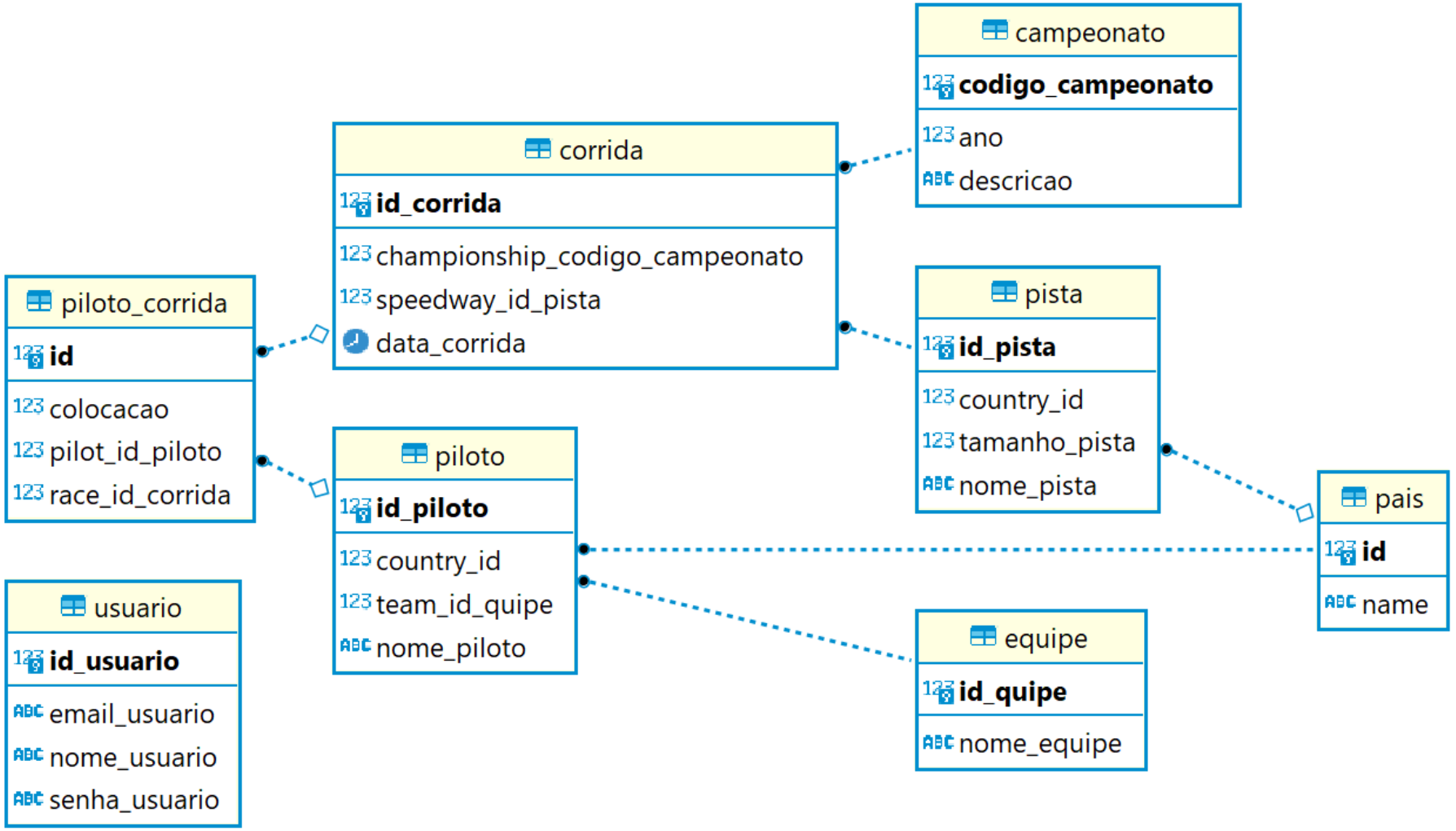


Testes unitários com JUNIT

Spring (campeonato fórmula 1)

unibave

Testes unitários com JUNIT



Testes de desempenho com JMeter

unibave

Testes de integração com Cypress

https://wcaquino.me/cypress/componentes.html

unibave

Ambiente

1 - Node JS - VSCODE - criar diretório para o projeto

2 - Iniciar o projeto `npm init -y`

3 - Instalar o cypress `npm install cypress --save-dev`

4 - Executar o projeto para criar a estrutura `npx cypress open`

Criar teste

1 - Diretório cypress/integration - extensão .spec.js ou .spec.ts

```
/// <reference types="cypress"/>
describe('Login', () => {
   it('should log in successfully', () => {
      cy.visit('https://www.example.com/login')
      cy.get('#username').type('myusername')
      cy.get('#password').type('mypassword')
      cy.get('button[type="submit"]').click()
      cy.url().should('include', '/dashboard')
   })
})
```

Principais Locators

ID:
```
cy.get('#myButton').click();
```

Classes:
```
cy.get('.myClass').click();
```

Atributos:
```
cy.get('[data-testid="myElement"]').click();
```

Tipo de elemento:
```
cy.get('input').type('Hello, Cypress!');
```

Texto:
```
cy.contains('Submit').click();
```

Posição:
```
cy.get('.myClass').eq(2).click();
// Clique no terceiro elemento com a classe 'myClass'
```

* contains
^ startsWith
$ endWith

Principais Assertivas

```
cy.should('be.visible')
```

Verifica se um elemento está visível na página.

```
cy.should('be.hidden')
```

Verifica se um elemento está oculto na página.

```
cy.should('have.text', 'texto esperado')
```

Verifica se um elemento tem o texto esperado.

```
cy.should('contain', 'texto esperado')
```

Verifica se um elemento contém o texto esperado.

unibave

Principais Assertivas

```
cy.should('have.value', 'valor esperado'):
```

Verifica se um elemento de entrada (input) tem o valor esperado.

```
cy.should('have.attr', 'atributo', 'valor esperado')
```

Verifica se um elemento tem um determinado atributo com o valor esperado.

```
cy.should('have.class', 'classe-esperada')
```

Verifica se um elemento tem a classe CSS esperada.

```
cy.should('have.css', 'propriedade', 'valor esperado')
```

Verifica se um elemento tem a propriedade CSS com o valor esperado.

Hooks

`before`
Executa uma vez, antes de todos os testes do arquivo.

`beforeEach`
Executa antes de cada teste.

`after`
Executa uma vez, depois de todos os testes do arquivo.

`afterEach`
Executa depois de cada teste

`beforeAll`
Executa uma vez, antes de todos os testes de todos os arquivos.

`afterAll`
Executa uma vez, depois de todos os testes de todos os arquivos.

Hooks

```
describe('Exemplo de testes', () => {
  // Executa uma vez antes de todos os testes
  before(() => {
    cy.visit('https://www.example.com');
  });

  // Executa antes de cada teste
  beforeEach(() => {
    cy.reload();
  });

  it('Teste 1', () => {
    // Código do primeiro teste
  });

  it('Teste 2', () => {
    // Código do segundo teste
  });

  // Outros testes...
});
```

unibave

Interagindo com TEXTOS e INPUT TEXTO

```
cy.get('.meu-elemento').should('have.text', 'Texto
esperado');

cy.contains('.meu-elemento', 'Texto parcial');

cy.get('#meu-input').type('Texto de exemplo');

cy.get('#meu-input').clear().type('Novo texto');
```

Interagindo com LINKs

```
cy.get('a').click();

cy.get('a').should('have.attr', 'href',
'https://www.example.com');

cy.get('a').invoke('removeAttr', 'target').click();
```

Interagindo com RADIO e CHECKBOX

```
cy.get('input[type="radio"]').check();

cy.get('input[type="radio"]').should('be.checked');



// Selecionar a opção de rádio com o valor "opcao2"
cy.get('input[type="radio"][value="opcao2"]').check();

// Desmarcar checkbox
cy.get('input[type="checkbox"]').uncheck();
```

unibave

Interagindo com COMBOBOX

```
cy.get('select').select('Opção 1');

cy.get('select').select('valor-opcao2');

cy.get('select').should('have.value', 'valor-opcao2');

//seleção múltipla
cy.get('select').select(['valor-opcao1', 'valor-opcao2']);
cy.get('select').should('have.value', ['valor-opcao1', 'valor-opcao2']);
```

unibave

Espera de processamento

```
cy.wait(2000); // Espera por 2 segundos (2000 milissegundos)

cy.wait(() => { // Condição para aguardar
   return cy.get('.elemento').length > 0;
});
```

```
// Configurar um timeout de comando de 10 segundos
cy.get('.elemento').timeout(10000);

cy.get('.elemento').should('be.visible', { timeout: 5000 });
```

Fixture

Permite carregar dados de um arquivo externo para serem utilizados nos testes `.json`, `.csv` ou `.xml`

```
cy.fixture('users.json').as('usersData');
```

Pode ser utilizado sempre que for necessário o dado do arquivo

```
cy.get('#meu-input').type(usersData.name);
```

Commands

Permite criar comandos personalizados usando o recurso de "commands" (comandos). < `support` -> `commands.js`>

```
Cypress.Commands.add('nomeDoComando', () => {
   lógica do comando personalizado aqui
});
```

```
cy.nomeDoComando();
```

```
//Buscar componente pela chave wicket

Cypress.Commands.add('getByWicket', (selector, ...args) => {
    return cy.get(`[wicketpath="${selector}"]`, ...args);
});
```